Characterization of Simulation Performance for Ethernet using SPaDES/Java Framework

Frujină Ionuţ-Andrei

ionut.frujina@gmail.com

A report presented in partial fulfilment for the degree of

Bachelor of Science

Supervisors:

Nicolae Ţăpuş

Dept. of Computer Science

University "Politehnica" of Bucharest

ntapus@cs.pub.ro

Teo Yong Meng

School of Computing

National University of Singapore

teoym@comp.nus.edu.sg

2004

Contents

List of Figures **4**

1 Introduction **6**

1.1 Simulation 6

1.2 Discrete event simulation 9

1.3 Parallel discrete event simulation 10

1.3.1 Conservative Protocols 12

1.3.2 Optimistic Protocols 14

2 SPaDES/Java **16**

2.1 Sequential Simulator 16

2.2 Time, Space and Strictness Analyzer 18

2.3 SPaDES/Java Simulator 19

3 Carrier Sense Multiple Access with Collision Detection **24**

3.1 Model 26

3.2 Implementation 29

4 Experimental results **32**

4.1 Parallelism requirement across layers 35

4.2 Memory requirement across layers 36

4.3 Strictness across layers 37

5 Conclusions **39**

Bibliography **41**

A Measurements **43**

B SPaDES Manual Page **45**

List of Figures

1.1 Ways to Study a System 8

1.2 Discrete-system state variable 9

1.3 Continuous-system state variable 9

1.4 Deadlock situation. Each process is waiting on the incoming link containing the smallest time-stamp (queue is empty) although there are events in other queues ready to process 13

2.1 SPaDES Measurements Tools 17

2.2 Three Layer Performance Analysis Framework 20

3.1 Model configuration for the distributed simulation 26

3.2 State diagram of the CSMA/CD Simulation 27

3.3 Simplified state diagram of the CSMA/CD Simulation 29

4.1 Relationship between frame size and null messages 34

4.2 Parallelism across layers 35

4.3 Parallelism at model layer 36
4.4 Memory requirement across layers 37
4.5 Strictness across layers 38

B.1 SPaDES/Java Man Page (1) 46
B.2 SPaDES/Java Man Page (2) 47

Chapter 1

Introduction

1.1 Simulation

Simulation is the process of designing a real system model, and conducing experiments on this model, in order to understand the behavior of the system or to evaluate different strategies for operation of the system.

A simulation model is the representation of a real system. This model is represented as a set of assumptions concerning the operation of the system. These assumptions are stated in logical, mathematical and symbolic relationships between the objects of interest in the system (entities). When a system model is developed and validated, it can be used to investigate various situations that can occur in the system. Any change of the real system can be simulated first in order to anticipate the changes that might appear. In addition, simulation could be used for studying systems before they are built, in the design state.

Standard simulation may not always be appropriate. Simulation cannot be used when the problem can be solved using common sense or if the problem can be

solved analytically. Also if it is easier to perform direct experiments, or the cost of the simulation exceeds the savings.

Simulation has many advantages, and even some disadvantages. For example, new policies, operating procedures, decision rules, organizational procedures, and so on can be explored without disrupting ongoing operations of the real systems. New hardware designs, physical layouts, transportation systems and so on can be tested without committing resources for their acquisition. Time can be compressed or expanded, allowing a speedup or slowdown of the phenomena under investigation. Another advantage of a simulation is the level of detail that you can get from a simulation. A simulation can give you results that are not experimentally measurable with our current level of technology. You can set the simulation to run for as many time steps you desire and at any level of detail you desire the only restrictions are your imagination, your programming skills, and your computer.

As a disadvantage we can consider that model building requires special training. There are also simulation errors. Any incorrect key stroke has the potential to alter the results of the simulation and give you the wrong results. Also, if two models are constructed by two different persons, they will have similarities, but they most certainly will not be the same. Another disadvantage is that simulation results may be difficult to interpret.

To model a system, it is necessary to understand the concept of a system. A system is defined as any organized assembly of resources and procedures united and regulated by interaction or interdependence to accomplish a set of specific functions. A system can be affected either by events occurring outside the system (system environment) or from the inside.

In order to understand a system, and to be able to analyze it, a number of

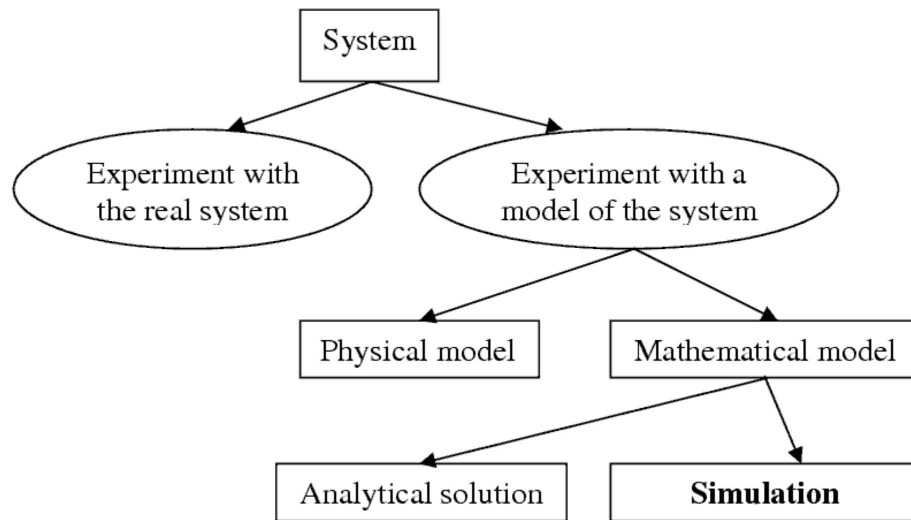

Figure 1.1: Ways to Study a System

terms must be defined. An *entity* is an object of interest in a system. An *attribute* is a property of an entity. An *activity* represents a time period of specified length. The collection of entities that compose a system for one study might be only a subset of the overall system for another study.[1] The *state* of a system id defined as that collection of variables necessary to describe the system at any time, relative to the objects of study. An *event* is defined as an instantaneous occurrence that may change the state of the system and schedule other events.

Systems can be classified as discrete and continuous. "Few systems in practice are wholly discrete or continuous, but since one type of change predominates for most systems, it will usually be possible to classify a system as being either discrete or continuous" [1]. A *discrete* system is one in which the state variables change only at a discrete set of points in time, while a *continuous* system is one in which the state variable change continuously over time.

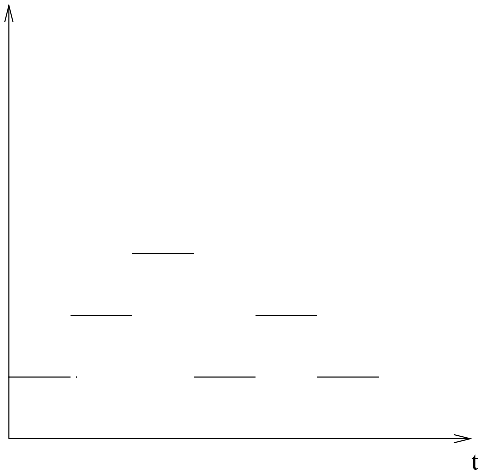

Figure 1.2: Discrete-system state variable

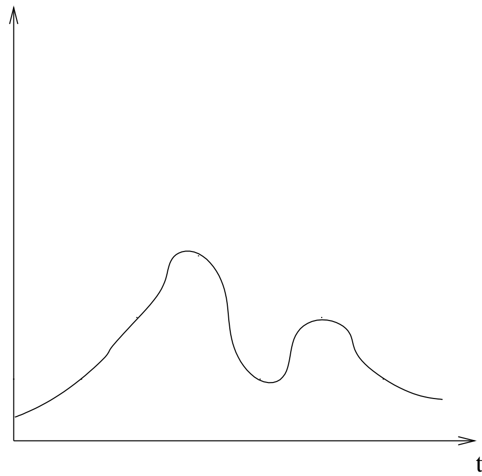

Figure 1.3: Continuous-system state variable

1.2 Discrete event simulation

Based on the characteristic of time, simulation models can be classified as static or dynamic. In *dynamic simulation* the system state changes with time, in contrast with *static simulation* where changes in the system state are independent of time. Dynamic simulation models may be further classified into *continuous* and *discrete* models, based on the characteristic of time. In continuous simulation, the system state changes continuously with time. A real system is often modeled using a set of differential equations. The system state in discrete simulation changes only at discrete points of time. Time can be advanced using a fixed time increment *time-stepped* or irregular time increment (*discrete event*). Simulation modeling can be done in three different ways: activity oriented, process oriented or event oriented. The most frequent used are event oriented and process oriented, but as the process oriented view is built on top of the event oriented, we will concentrate on this.

1.3 Parallel discrete event simulation

Parallel discrete event simulation refers to the execution of a single discrete event simulation program on a parallel computer. Parallel discrete event simulation has attracted a considerable amount of interest lately because many simulation consume enormous amount of time on sequential machines. It is also interesting because it contains substantial amount of parallelism, but it is difficult to parallelize.

As we are especially concerned with the simulation of asynchronous systems, where evens are not synchronized by a global clock, but rather occur at irregular time intervals, parallelization techniques based on lock-step execution using global simulation clock perform poorly or require assumptions in the timing model that compromise the fidelity of the simulation [2]. So, concurrent execution of events at different point in simulated time is required, but this introduces synchronization problems.

It was proposed using dedicated functional units to implement specific sequential simulation functions [3, 4]. This model can provide only a limited amount of speedup. Others proposed to use a hierarchical of the simulation model to allow an event consisting of several sub-events to be processed concurrently [5, 6]. A third alternative of execution is to execute independent, sequential simulation programs on different processors[7, 8]. it is useful for largely stochastic simulations or for simulations that must be done on large numbers of different parameters.The major drawback of this alternative is that each processor must contain sufficient memory to hold the entire simulation.

In order to describe the problems of the parallel discrete event simulation, I

will examine the operation of a sequential discrete event simulator. Sequential simulators typically have three data structures: *the state variable* that describes the state of the system, an *event list* containing all pending events(each event has a *time-stamp*) that have been scheduled, bat have not yet taken effect, and a *global clock* variable to show how far the simulation has progressed. The principle is that the event with the minimum time-stamp is removed from the list, the simulation clock is advanced, and the event handler (some simulator code) is executed. This is done until a stopping condition is met.

If we try to parallelize, it means more events are picked from the event list, one for each processor. But if one event for example modifies some state variables that another event uses, we have a *causality error*. So, certain *sequential constrains* must pe maintained in order for the computation to be correct.

The system being modeled, usually referred to as the *physical system*, is viewed as being composed of some number of *physical processes* (PP) that interact at various points in simulated time. For example, in a communication network simulator, the physical processes might be switching centers that interact by transmitting data over communication lines. Parallel discrete-event simulation uses this information to partition a simulation model into smaller components called *logical processes*(LP). Parallelization in simulation is done by simulating logical processes concurrently.All interactions between physical processes are modeled by time-stamped event messages sent between the corresponding logical processes. Each logical process contains a local clock that denotes how far the process has progressed and portion of the state variable of the corresponding physical process. There are two potential benefits of implementing parallel simulator: reduced execution time and facilitating the execution of larger models. In simulation, *local*

causality constraint imposes that if event *a* happens before event *b* and both events happen at the same logical process, then *a* must be executed before *b*. Parallel simulation must adhere to local causality constrain to produce correct simulation results. Based on how local causality is maintained, parallel simulation protocols are grouped into two main categories: *conservative* and *optimistic*.

1.3.1 Conservative Protocols

Conservative protocols do not allow any local causality constraint violation throughout the duration of the simulation. Optimistic protocols allow local causality constraint violation, but provide mechanisms to rectify it. A more detailed taxonomy of simulation mechanisms is described in [9].

To avoid the occurrence of straggler events which indicate local causality constraint violation, Chandy, Misra and Bryant proposed building a static communication path for every interacting LP [10, 11]. In the conservative mechanisms, if a process contains an unprocessed event E_1, with time-stamp T_1(the smallest) and the process can determine that it is impossible for it to later receive another event with time-stamp smaller than T_1, E_1 can be processed because the local causality constraint is not violated.

If a process does not contain any safe events, it blocks (deadlocks might occur). In order to determine when it is safe to process a message, it is required that the sequence of time-stamps on messages sent over a link be non-decreasing. The main idea is to statistically specify the links that indicate which process can communicate with which other process. Then every process has a queue for each incoming link from which it selects the smallest clock, or if the queue is empty, it blocks. This can lead to deadlocks. The solution is to use null messages, for example a message

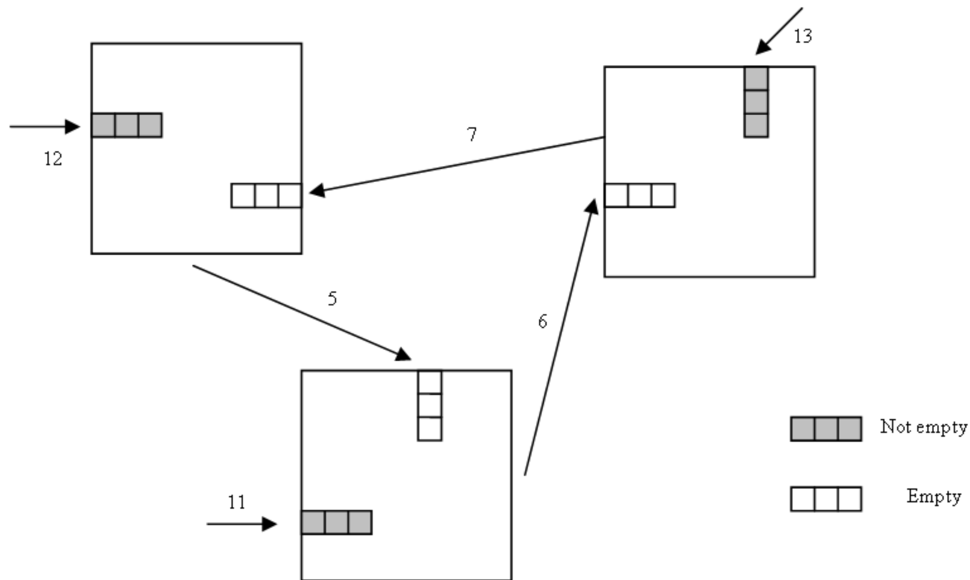

Figure 1.4: Deadlock situation. Each process is waiting on the incoming link containing the smallest time-stamp (queue is empty) although there are events in other queues ready to process

with time stamp T_{null} is sent from LP_a to LP_b is a promise by LP_a that it will not send a message to LP_b with time stamp smaller then T_{null}. The condition for this scheme to be valid is to have a cycle of links with the same link clock time.

There are some improvements of this algorithm, like *demand driven protocol* [12], where LP sends null-messages only on demand. There is also the *flushing protocol* [13], when a null message is received, an LP flushes all null messages that have arrived but not been processed.

Also there are other conservative protocols such as *bounded-lag protocol* and *conservative time window*. The main idea of this protocols is that they start by determining safe events, then execute them. A barrier is activated for synchronization between identification and execution of events. This protocols are suitable especially for shared-memory architecture because of the synchronization mechanisms. Another approach is exploiting the lookahead (the ability to predict what

will happen, or more importantly what will not happen in the simulated future). It is used in deadlock avoidance or in deadlock detection/recovery.

The performance of conservative mechanisms is usually poor, due to failure of algorithms to exploit parallelism, rather then overheads, associated with the implementation of the algorithms. Usually conservative algorithms are better suited for large problems. If the lookahead is small, conservative algorithms perform very poor, and minor changes to the application have huge impacts on performance.

1.3.2 Optimistic Protocols

There is also an alternative to the conservative protocols, the optimistic approach. These protocols detect and recover causality errors, they do not strictly avoid them. Usually every event that is ready to execute is executed. They detect when an error has occurred, and invoke a procedure to repair it. The advantage of these protocols is that parallelism can be exploited in a case where causality might occur. Dynamic creation of processes is also possible, unlike conservative methods. *Time Wrap Protocol* [14, 15] is one of the best optimistic protocols(an error occurs if an event has a time-stamp smaller then the process clock. The event causing a rollback is called a straggler. An event may do two things that have to be rolled-back: modify a state variable, or send a event message to another process. The rollback is done with *anti-messages*. In Time Wrap, the smallest time-stamp among all unprocessed events is called /emphglobal virtual time, and no rollback is performed before this time.

As an example of algorithms we have *lazy-cancelation*, *aggressive-cancelation* which repair the damage caused by an incorrect computation rather than completely repeat it, respectively roll-backing it immediately. Of course performance

is degraded if too many rollbacks must be executed (overhead). There is also *lazy reevaluation (jump forward)* which is similar to lazy cancelation, but deals with state vectors rather then messages. When it is clear that the execution of an event will be the same, do not execute it again, just jump forward.

There are some critiques for the optimistic algorithms. First of all, it is possible that most of the time to be used on incorrect computations. The further the incorrect computation go into the simulated future, the lower the priority (smaller time-stamps have higher priority). another disadvantage is the need to periodically save the state of each logical process(overhead) and also the use of memory which is several times greater than conservative algorithms Finally, erroneous computation may enter infinite loops and also optimistic protocols are much more complex to implement.

Chapter 2

SPaDES/Java

The framework for characterizing the simulation performance is called SPaDES/Java. There is a set of measurements tools, that measures the performance at three different layers: the physical system layer, the simulation model and the simulator layer. Here is a scheme of the measurements tools:

The order in which the events are executed depends on the different event orders. The degree of dependency between the events is affected by the strictness of the ordering rules. So there was proposed a relation *stricter* and a measure called *strictness* for comparing the event dependencies. The measurements at the physical system layer are Π^{prob} and M^{prob}; at the simulation model layer the measurements are Π^{ord} and M^{ord} and finally at the simulator layer we measure Π^{sync} , M^{sync} and M^{tot}. Also strictness can be measured at all the three layers.

2.1 Sequential Simulator

The sequential simulator simulates a problem, the physical system. It is also used to measure event parallelism, Π^{prob} and memory requirement, M^{prob}. The simulator

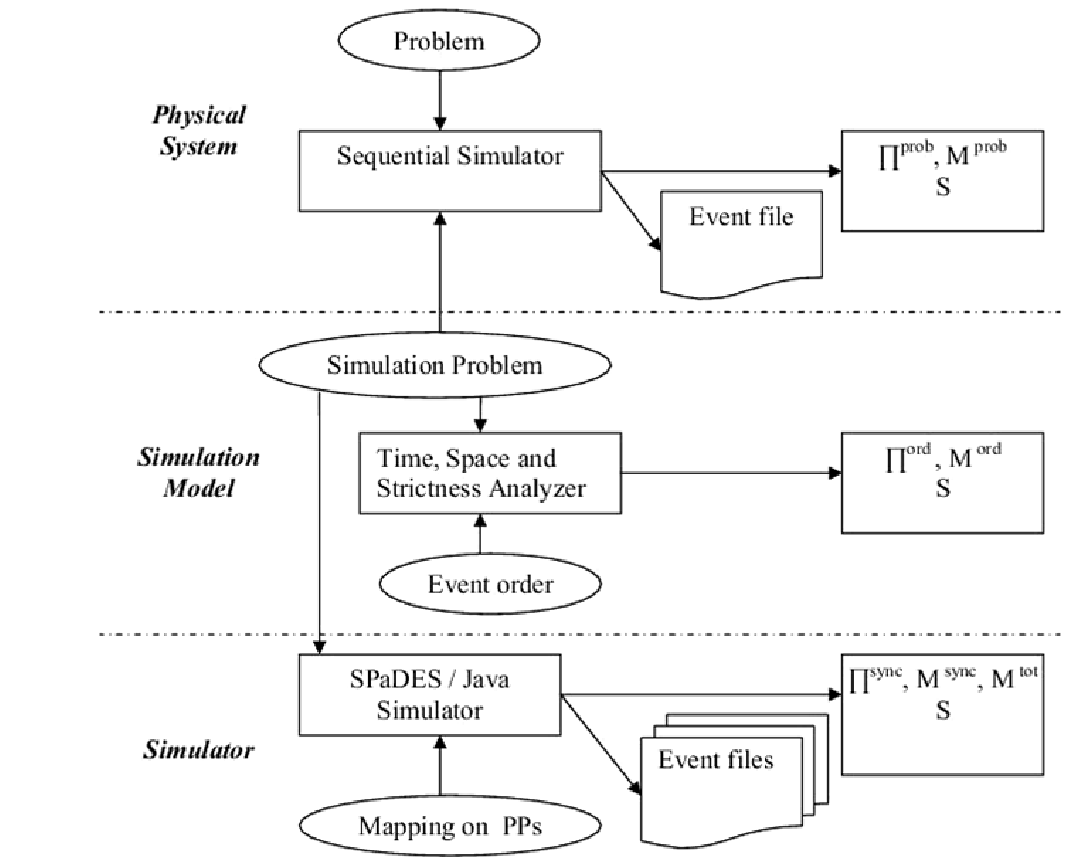

Figure 2.1: SPaDES Measurements Tools

produces a log file that contains the order in which the events are executed. Event parallelism, Π^{prob}, is measured in the simulator, based on the number of events. The memory requirement,M^{prob} is also measured, based on the maximum queue size. The log file generated is used by the Time, Space and Strictness analyzer to simulate and measure the performance of different event orderings as detailed later.

The algorithm for the simulator is basically the following:

1. *while (stopping condition has not been met)*
2. *remove event e with the smallest timestamp from FEL*
3. *simulation_clock = e.timestamp*
4. *execute (e)*
5. *add the generated events to FEL*

2.2 Time, Space and Strictness Analyzer

Time Space and Strictness Analyzer is used to simulate different event orderings and to measure the event parallelism at the simulation model layer, Π^{ord}, and also the memory requirement M^{ord} and the strictness , S , at all the three layers.

In order to measure Π^{ord} and M^{ord}, the Time, Space and Strictness Analyzer needs the log file generated by the sequential simulator. Then an event ordering can be specified. The log file contains every event executed by the sequential simulator, and also the dependencies between events. Having an event ordering, the Time, Space and Strictness Analyzer simulates the execution of events, measuring also Π^{ord} and M^{ord}.

In order to measure the strictness at all the three layers the Time Space

and Strictness Analyzer needs the log file form the sequential simulation (for the first two layers, physical and simulation model) and also the log files from the SPaDES/Java Simulator(for the simulator layer).

The method used to determine the strictness of event ordering is the following: a fixed number of events is read form the log file, and strictness is measured based on the given event ordering. And so on for all the events. Event ordering strictness is determined by summing up the strictness at each step, and dividing by number of steps. This method was chose because measuring the strictness of event orderings with a large number of events not efficient (computationally).

2.3 SPaDES/Java Simulator

SPaDES/Java is a parallel simulator library that supports event oriented worldview. It supports a parallel simulation based on CMB protocol with demand driven optimization [12]. It is used to measure effective event parallelism, Π^{sync} and memory requirement for overhead events, M^{sync} and the total memory requirement M^{tot}. The simulator produces a log file for each physical processor that contains the order in which the events are executed on each physical processor. Effective event parallelism, Π^{sync}, is measured in the simulator, based on the number of events and the simulation time. The memory requirement for overhead event,M^{sync} is also measured, based on the maximum size of the data structure used to store the overhead events. The log file generated is used by the Time, Space and Strictness analyzer to measure strictness of event ordering at the simulator layer.

SPaDES/Java uses RMI API in Java for message passing needed to implement

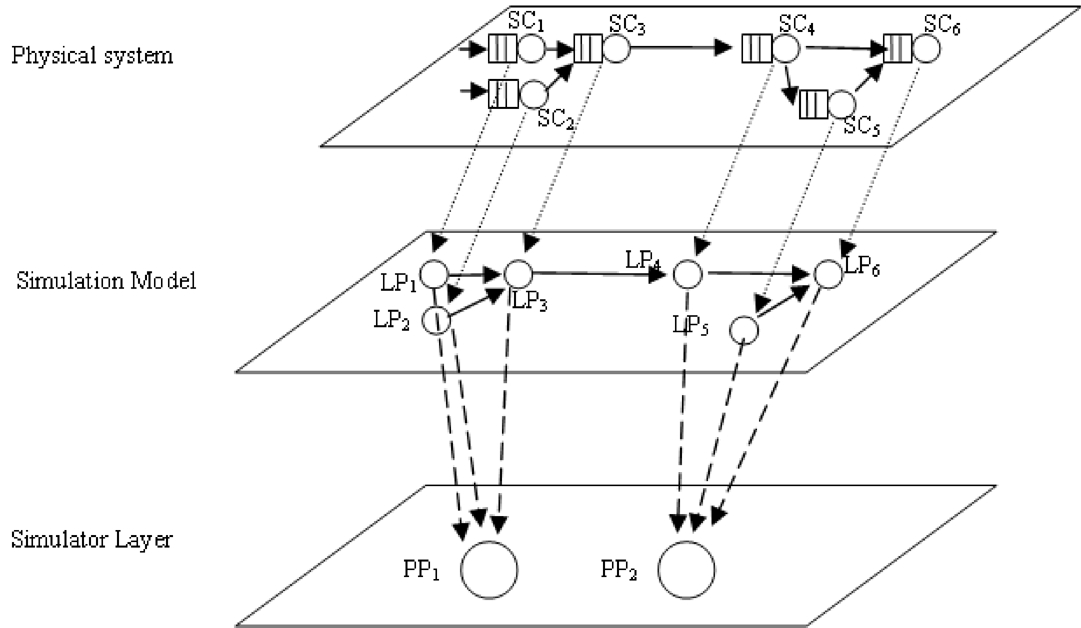

Figure 2.2: Three Layer Performance Analysis Framework

the parallel event synchronization. There is a communication manager (CommManager) that manages all the message passing activities between the processors. During the simulation, all the LPs on the same host are managed by one CommManager running on that host. The location information of all the LPs is stored in a table of the CommManager. When an LP is to send out a message to a receiver LP, it will first check the location of that receiver LP according to the table in its local CommManager. If the receiver LP is on the same host as the sender LP, then the sender LP will just call the pass the message to the receiver LP through shared memory. If the receiver LP is on a remote host, the sender LP will call the CommManager to send the message to the CommManager running on that remote host through RMI. In this case, the receiver CommManager will receive the message and route it to the proper receiver LP through shared memory. In SPaDES/Java simulation environment, an LP has the following data structures for

parallel simulation based on conservative event synchronization protocols :

- an event list containing all pending events that have been scheduled, but have not yet taken effect
- a local clock variable to denote the local virtual time (LVT) of an LP
- an array of input channels each of which stores the timestamp of the last incoming event message from one LP connected to this LP, and
- an array of output channels each of which stores the timestamp of the first outgoing event message to one LP connected to this LP and the reference to the first outgoing event message.

The event list is a vector storing event messages. When an event message arrives at an LP, it will be stored into the event list to be processed and its timestamp will also be recorded in the corresponding input channel. The LP's executive routine advances the global clock to the timestamp of the next process in the event list according to the null message algorithm. It then removes this event, executes it, and sends out any new outgoing event generated from executing this event. Then the executive cycle loops again for the entire duration of the simulation.

The event synchronization protocol used in SPaDES/Java is the conservative null message protocol. The algorithm for the traditional null message protocol, developed by Chandy, Misra and Bryant is described below :

CHANDY-MISRA-BRYANT()

1 while simulation is not over

2 do

3 wait till the FEL contains at least one message;

4 M <- remove smallest timestamped message from FEL;

5 clock <- time stamp of M;

6 execute(M);

7 send null message to neighbouring LPs with

8 timestamp equal to lower bound on timestamp of

9 future messages (clock + lookahead);

Our implementation of SPaDES/Java adopts a modified version of the null message protocol. Instead of adopting the greedy approach of transmitting null messages every time the LP processes an event message, null messages are only sent whenever the LP becomes blocked on at least one of its input channels, waiting for messages to arrive from them. The modified algorithm is described below :

IMPROVED-NULL-MESSAGE-ALGORITHM()

1 Initialization:

2 Initialize the timestamps of all input channels to LVT;

3 Loop:

4 M ¡- remove smallest timestamped message from FEL;

5 if M != null and t ¿= timestamp of M

6 then execute(M);

7 else

8 for all output channels(OP) i

9 do

10 if timestamp of OP_i *= null*

11 then send null messages with

12 timestamp = t + lookahead to

13 the LP connected to OP_i*;*

14 else

15 send null messages with

16 timestamp = timestamp of OP_i *to*

17 the LP connected to OP_i;

18 end for

19 wait until there is an incoming message

20 end loop

Chapter 3

Carrier Sense Multiple Access with Collision Detection

The most commonly used medium access control protocol for Ethernet is Carrier Sense Multiple Access with Collision Detection (CSMA/CD) [16]. Under this protocol, a station that attempts to transmit must listen to the medium first to determine whether the medium is in use or not. If the medium is in use, then the station must wait, otherwise it might transmit. It is possible that two or more stations transmit almost the same time so that all of the sense that the medium is idle. If this happens, there will be a collision, ant the frame being sent will be garbled. Therefore, it is important for a station to be able to detect a collision. To account for this, during transmission a station has to listen to the medium whether one or more other stations are transmitting their frames for up to two propagation delay time. If collision is detected during the transmission, then the station will transmit a brief jamming signal to assure that all stations which are involved in the collision must wait for a random amount of time before attempting to retransmit

their frames (back off). The simulation model adopted for the Ethernet is the one developed by Wang and Keshav [17].

Ethernet refers to a family of Local Area Network (LAN) multiple access protocols that vary in details such as bandwidth, collision detection mechanisms etc. In this paper we use Ethernet to mean an unslotted, 1-persistent, carrier-sense multiple access method with collision detection (CSMA/CD) and binary exponential back-off. Here is a brief review of the CSMA/CD protocol. Assume that n stations attach to the Ethernet link (Figure 3). A station senses the medium before sending a packet and sends the packet immediately after the medium is idle. If a collision is detected while sending a packet, the station sends out a jam signal and exponential back-off scheme is employed. Upon a collision, the station waits for a random time chosen from the interval [0, 2 * max propagation delay] before retransmitting the collided packet. If retransmission fails, the station backs off again for a random time chosen from the interval with double length of the previous one. Each subsequent collision doubles the back-off interval length until the retransmission succeeds (the back-off interval is reset to its initial value upon a successful retransmission of packet). If the back-off interval becomes too large (e.g. after 16 retransmission), the packet is dropped and the back-off interval is reset.

Most existing CSMA/CD simulation model the transmission medium as an active centralized entity. This entity determines the exact moment in time at which each station knows that a packet was placed on the medium or a collision has occurred. It is very difficult to determine this times because many packages can be placed on the medium at different points at very close moments in time. In order to determine very accurate this times signal's electromagnetic propagation should

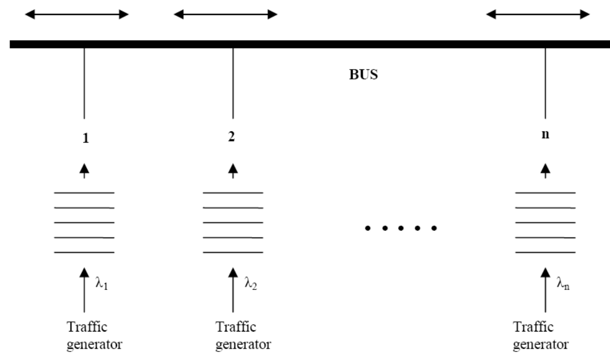

Figure 3.1: Model configuration for the distributed simulation

be simulated on the medium But this is very difficult both algorithmically and to implement. After many attempts of creating a precise simulation of CSMA/CD, an alternative approach was chosen. They considered that the medium is passive. But each station on the Ethernet acted as a router, forwarding the packages from one station to another. An idle station that receives a packet changes its status to busy. If a packet arrives at a station that has the status set to busy, a collision occurs, and the station broadcasts a jam signal to all the other stations. In this approach, stations collaborate for simulating the medium. This makes the simulation a lot easier to implement and to validate.

3.1 Model

Wang and Keshav [17] modeled CSMA/CD using the station state diagram shown in Figure 3.2. It can be seen that the medium is not modeled as an active entity. Instead, stations exchange data, jam, and collision messages as would happen in an actual Ethernet. Each simulated station is responsible for actions such as packet transmission/retransmission,collision detection, signaling. A simulated station can

be in one of the seven states: idle, *sending*, *receiving*, *wait for back-off end and jam end*, *wait for jam end*, *wait for back-off end*, and *receiving and wait for back-off end.*

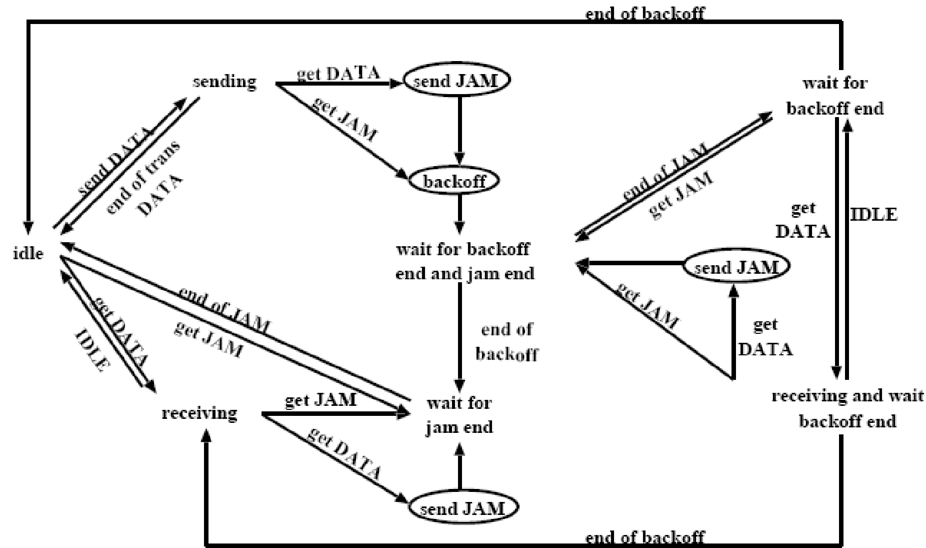

Figure 3.2: State diagram of the CSMA/CD Simulation

In our implementation, we use the same model but a little bit simplified. There are only four states in which each station can be at a certain moment in time. This states are *sending*, *receiving*, *idle* and *wait for back-off end.* Packet propagation on the Ethernet is simulated as consecutive propagation by the intermediate stations towards the destination. In this model we use six events in order to model the frames moving from one station to another.

Frame arrival which represents the arrival of a frame at the MAC layer of the station. If the station is not idle the frame will be buffered until the status of the station is idle. If the station is idle, the frame transmitted.

Begin transmit data which appears when the station starts transmitting the frame to its neighbors. When *begin transmit data* occurs the station switches from

the *idle* status to the *sending* status.

End transmit data which appears when the station finishes transmitting the last bit of the frame. When *end transmit data* occurs the station switches from the *sending* status to the *idle* status.

Begin receive data which appears when a station receives the first bit of a frame sent by its neighbor. When *begin receive data* occurs, if the status of the station was *idle* it switches to *receiving*. If the status was *sending* than a collision occurs tat means that the status is switched to *wait for back-off end*.

End receive data occurs when the station receives the last bit of a frame sent by its neighbor. When *end receive data* occurs the station switches from the *receiving* status to the *idle* status.

Finally, *end back-off* which appears when the period of time required for waiting expires. The station switches to the previous state and tries to retransmit its corrupted frame until the maximum retransmission is reached and the frame will be dropped.

In Figure 3.3 the diagram of our model is shown.

We choose the 10BASE5 Ethernet specification and the following assumptions and parameters: the time between packets is uniformly distributed, there is one packet buffer at each station of finite size and the spacing between station is the same. So, the assumptions that were made for this model are the *frame size*, which can se set between the minimum value of 64 bytes and the maximum of 1518 bytes, the *end to end propagation delay* which is set to 30μs and *node to node propagation delay* which is set to 1μs. There is also the *jamming signal size* which is set to 4 bytes, the *LAN speed* of 10Mbps and the *maximum buffer size* at each station which is set to 8 packets.

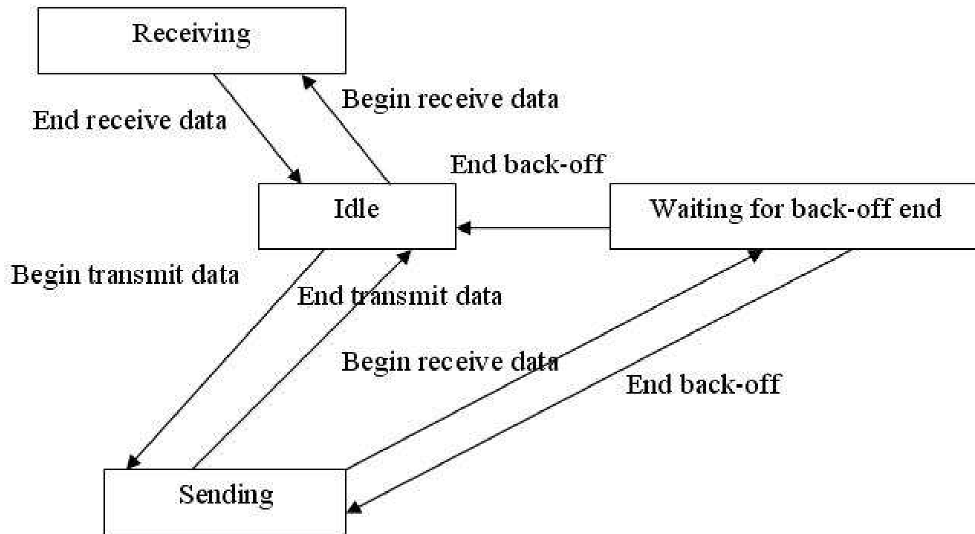

Figure 3.3: Simplified state diagram of the CSMA/CD Simulation

3.2 Implementation

The SPaDES/Java simulator offers a template for implementing different problems. The CSMA/CD protocol is implemented using this template based on the model described in the previous section. The classes that need to be extended are **SimJavaParam** which contains the description of the parameters, the **SimJavaSimulator** which is responsible for starting the simulator, the **SimJavaKernel** which is the class that executes the events, and at least two instances of **SimJavaEvent** in order to have different types of events.

Here are the files that were implemented:

CSMAArrival.java - extends SimJavaEvent

CSMABeginRxData.java - extends SimJavaEvent

CSMABeginTxData.java - extends SimJavaEvent

CSMADef.java- interface

CSMAEndBackOff.java - extends SimJavaEvent

CSMAEndRxData.java - extends SimJavaEvent

CSMAEndTxData.java - extends SimJavaEvent

CSMAKernel.java- extends SimJavaKernel

CSMAParam.java - extends SimJavaParam

CSMASimulator.java - extends SimJavaSimulator

CSMAStateVar.java - extends SimJavaStateVar

First we have the class **CSMAParam** which extends the **SimJavaParam** where parameters for this particular problem are described. The parameters that can be set in this problem are *frame size*, *end to end propagation delay*, *node to node propagation delay*, *jamming signal size*,*LAN speed*,*maximum buffer size*, *network load* and the *packet transmission time*. The general parameters such the *problem size* or *stopping condition* are stored in the base class.

Then there is the **CSMAStateVar** which extends the **SimJavaStateVar** where state variables are stored. This class contains only the state variables specific for this problem, general purpose variables such as the queue are stored in the base class.

The main class, the one that starts the simulator is **CSMASimulator** which extends the **SimJavaSimulator**. Here the configuration file is read and the simulator is initialized. Also the the communication matrix is initialized.

The **CSMAKernel** class is the one in charge of the execution of events. One instance of the kernel is running on each physical processor (if the parallel version of the simulator is used). The kernel receives the parameters form the simulator and based on this parameters and using the Java RMI it communicates with the other instances of kernels on other physical processors. The kernel also generates

the report files for each of the physical processors. If the sequential version of the simulator is used, all the logical processes are mapped on the same physical processor.

There are six types of events defined for the CSMA/CD implementation: *Arrival*, *BeginRxData*, *BeginTxData*, *EndRxData*, *EndTxData* and *EndBackOff*.

CSMAArrival is the event that appears when a frame arrives. If the status of the current logical process is *idle* then *BeginTxData* is generated at the current process, else the new package is queued. Also a new arrival is scheduled.

When **CSMABeginTxData** is executed, the logical process sends a **CSMABeginRxData** to all other logical processes and generates for itself a **CSMAEndTxData**. The status of the logical process switches to *sending*.

When **CSMABeginRxData** is executed, if the status was *idle* the status changes to *receiving*, otherwise a collision occurs and the **CSMAEndRxData** is sent to all the other logical processes. Then a **CSMAEndBackOff** is generated, and if the number of back-off attempts is reached the total number of failed packages is incremented, otherwise, this the number of retries is incremented.

After **CSMAEndTxData** occurs, the **CSMAEndRxData** event is sent to all the other logical processes and also o a new **CSMABeginTxData** is generated on the current process.

When the **CSMAEndRxData** is executed, if the status is *receiving*, the **CSMABeginRxData** is sent to the other logical processes.

Finally, when **CSMAEndBackOff** is executed, **CSMABeginRxData** is sent to all logical processes, and **CSMAEndTxData** is generated for the current logical process.

Chapter 4

Experimental results

Having the framework for characterizing the simulation performance from the physical system layer to the simulator layer and the set of measurement tools we can apply the framework in order to study the performance of the simulation.

There is a set of experiments for testing the framework. The experiments required for measuring the performance at the physical system layer and the simulation model layer are conducted on a single processor machine. The experiments at the simulator layer, those that use SPaDES/Java parallel simulator are conducted on a cluster of computers, connected via Gigabit Ethernet. Each node is a dual 2.8GHz Intel Xeon with 2.5 GB of RAM.

The objective of this paper is to study the performance of Ethernet simulation. We will concentrate in our experiments on the parallelism on different layers in the simulator.

Parallelism at the three different layers can not be compared directly. At the problem layer, the parallelism is measured in events per μs. At the simulation model layer the event parallelism is measured in events per timestamp, and finally

at the simulator layer it is measured in events per ms.

In order to be able to compare the three types of event parallelism a normalization is required. The base of normalization was chose the simulation model layer, so the measurement unit will be events per timestamp. At the physical system layer, the conversion is done using this formula:

$$\Pi_{norm}^{prob} = \Pi^{prob} * \frac{D^{prob}}{D_{ts}^{ord}} \tag{4.1}$$

where Π^{prob} is the one measured at the problem layer, D^{prob} is the total duration at the same layer and D_{ts}^{ord} is the total duration measured at simulation model layer using the timestamp ordering.

The parallelism at the simulation model layer is the base of normalization, so only the parallelism value at the simulator layer must be normalized. This can not be done easily, by a formula, and it is done by the *Time, Space and Strictness Analyzer*. The normalized parallelism at the simulator layer is determined by counting the actual number of steps required by the simulator to run. Then the normalized parallelism is computed as follows:

$$\Pi_{norm}^{sync} = \frac{\#events}{\#steps} \tag{4.2}$$

In our experiments we are interested in parallelism loss/gain across layers. But the CSMA/CD problem has a lot of parameters that can affect the parallelism. Some of them have a well known impact on performance, so in our experiments are fixed.

For example there is the frame size. It can vary form 64 to a maximum of 1512 bytes. If we increase the frame size, the number of packets will decrease, so

the number of events will decrease also. This happens because a larger frame size implies that the transmission time (the time to complete frame transmission) is longer. Therefore, when a station has executed *BeginTxData*, the station can only execute event *EndTxData* after executing at least a number of null messages (time to complete transmission divided by propagation delay between two stations). So if the frame size is larger, there will be a lot more null messages executed(Figure 4.1).

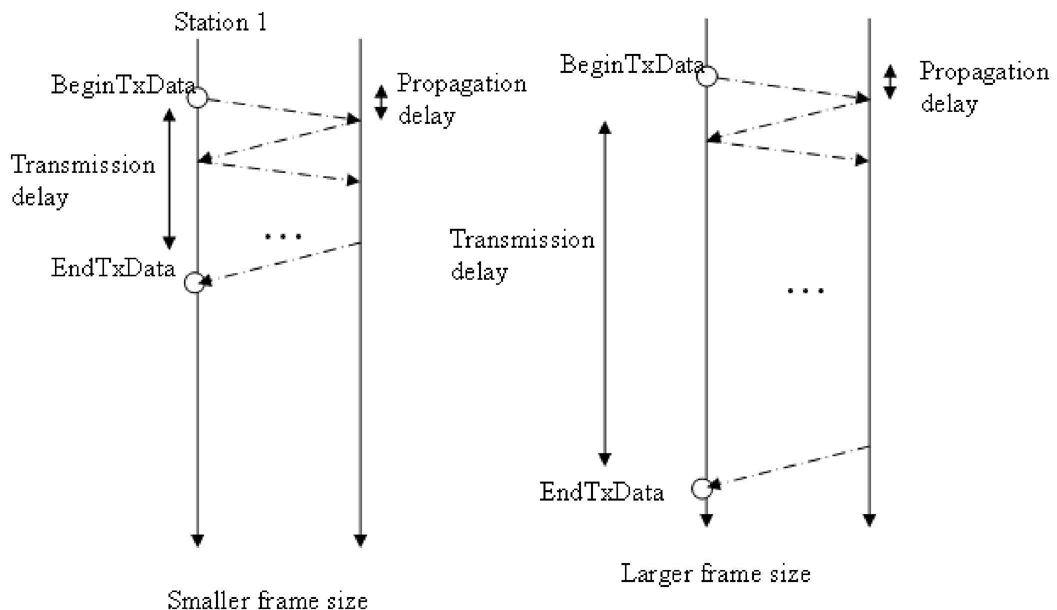

Figure 4.1: Relationship between frame size and null messages

The parameters that I chose to vary are the problem size (number of stations in the simulation) and the number of physical processors. Of course, at the simulation model layer, I have used different orderings. Here are the results for the CSMA/CD simulation when varying the number of stations from 24 to 96.

4.1 Parallelism requirement across layers

As it can bee seen in Figure 4.2 the parallelism of the problem is relatively small. In our example the number of physical processors at the simulator layer varies from two to eight.

All the parallelism values in Figure 4.2 and Figure 4.3 are measured in events per time stamp, values corresponding to the normalized parallelism.

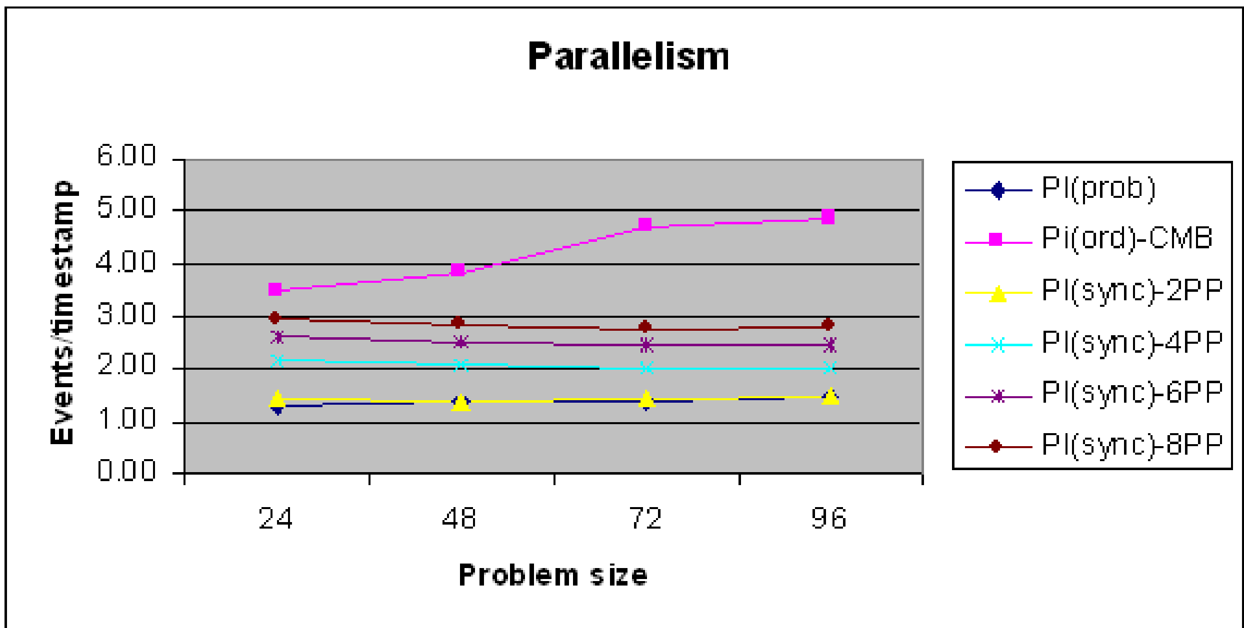

Figure 4.2: Parallelism across layers

As the number of physical processors increases, so does the event parallelism. The parallelism value grows towards the value obtained at the simulation model layer, using the CMB protocol (because the simulator uses the CMB protocol).

Although the increase of number of physical processors is linear the parallelism increases logarithmic, because of the null message density, which increases with the number of physical processors (more processors means more communication

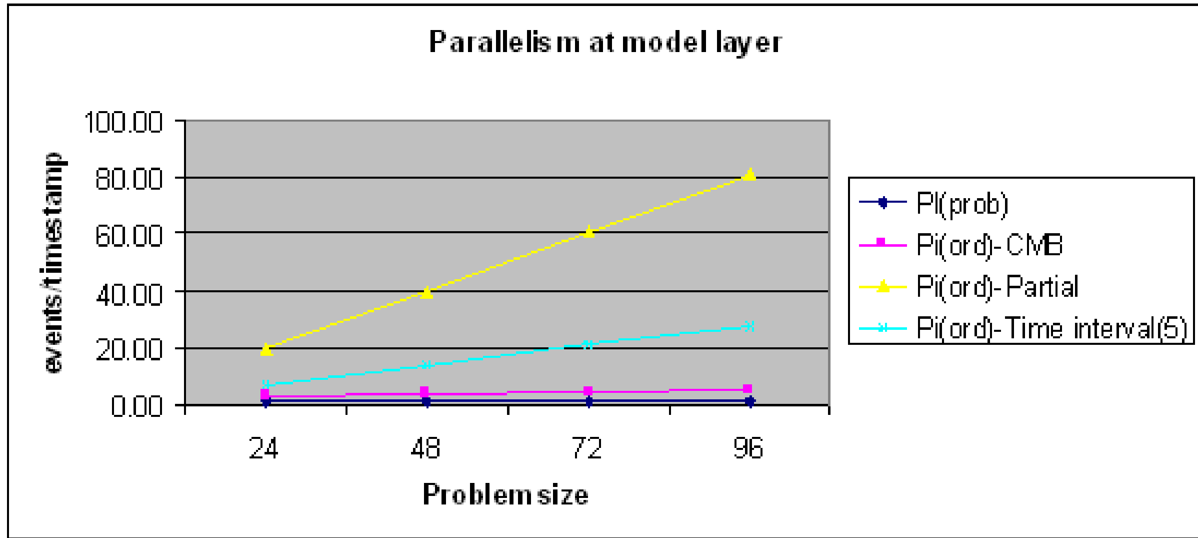

Figure 4.3: Parallelism at model layer

between them).

Considering only the parallelism at the simulation model layer (Figure 4.3)we can see that different ordering could produce more parallelism.

By far the best result is obtained using the partial ordering. But partial ordering means less constrained in event dependencies, so a lot more difficult to implement at the simulator layer. But as our simulator is implemented using the CMB ordering this is the value of parallelism that we should refer to.

4.2 Memory requirement across layers

As we can see in Figure 4.4 the memory required for synchronization (Π_{sync}) grows significantly with problem size. This happens because the memory is derived from the queue size on each station.

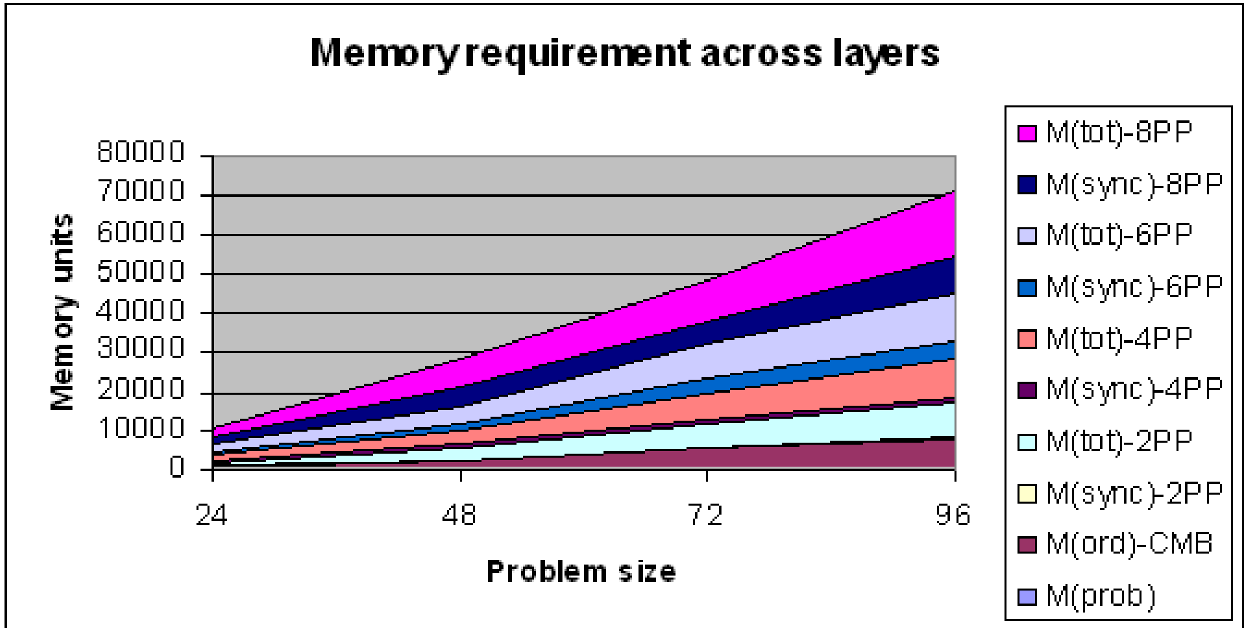

Figure 4.4: Memory requirement across layers

As we increase the problem size, Π_{sync} becomes the dominant factor in Π_{tot}, and this happens because the null message ration increases dramatically towards 1, so the memory structures required to store the null messages are larger.

4.3 Strictness across layers

Finally we come to the measurement that is proposed in the SPaDES/Java Frame Work, strictness. As mentioned above, strictness is a measure introduced in order to determine event dependencies.

As we can see in Figure 4.5 strictness of CMB event ordering is very high so it does not yield enough parallelism. There fore CMB event ordering is not suitable for Ethernet simulation. Another ordering such ar partial ordering should considered. One of the reasons why strictness is so poor for Ethernet simulation is

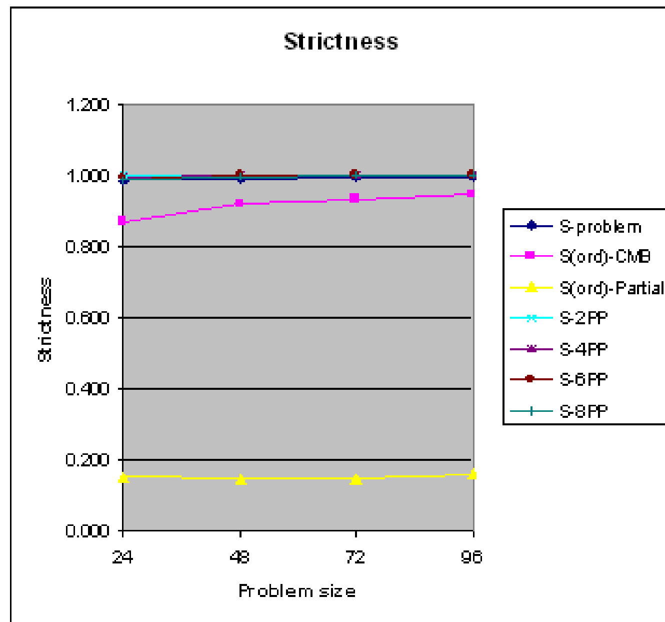

Figure 4.5: Strictness across layers

the big difference between frame transmission time and propagation delay, which results in poor lookahead.

Chapter 5

Conclusions

We have a framework for characterizing the simulation performance from the physical system layer to the simulator layer and the set of measurement tools(SPaDES/Java Framework).

We applied the framework in order to study the performance of a particular simulation problem. I chose the CSMA/CD problem (Ethernet) to be simulated.

In order to model the CSMA/CD problem I used a simplified version of Wang and Keshav [17] model for ethernet simulation. The main feature of this model is that the medium of Ethernet is not seen as active, it is considered passive, so each station on the Ethernet acted as a router, forwarding the packages from one station to another. This made the simulation a lot easier to implement and to validate.

Based on the experimental results we concluded that Ethernet Simulation is not suited for simulation because the parallelism of the problem can not be fully exploited because of the CMB implementation of the Simulator which introduces a large overhead and a high event dependency.

Another disadvantage of Ethernet simulation is that events in real world (problem layer) happen at a rate of a few billions per second (ns) while at the simulator layer, events cannot be executed at a rate higher than a few million per second (μs).So simulation is actually slower than the real world.

In order to make Ethernet suitable for simulation, another event ordering should be considered (such as partial).

Another approach I have tried was to use the Fast Ethernet (100Mbps) in my simulation. The difference between Fast Ethernet and Ethernet is that the transmission time drastically reduces, reducing also the null message ratio.

Bibliography

[1] Law, A. M., and W.D. Kelton [2000], *Simulation Modeling and Analysis*, 3d ed., McGraw-Hill

[2] Fujimoto, R.M. [2000], *Parallel and Distributed Simulation Systems*

[3] Comfort, J.C. [1984], *The simulation of a Master-s;ave Event Processor*, pp117-124

[4] Davis, C.K., Sheppard, S.V., and Livey, W.M. , [1988], *Automatic Development of Parallel Simulation Models in Ada* pp339-343

[5] Conception, A.I., [1989], *A Hierarchical Computer Architecture for Distributed Simulation*, 311-319

[6] Zhang, G. and Zeigler, B.P. [1989] *DEVS=Scheme supported mapping of hierarchicl models onto multiple processor systems*, pp64-69.

[7] Biles W.E., Daniels, D.M. and O'Donnel, T.J., [1985], *Statistical considerations in simulation on a network of microcomputers*, pp 388-393

[8] Heidelberger, P., [1986], *Statistical analysis of parallel simulations*, pp 290-295

[9] Reinolds, P.F., Jr. [1988], *A spectrum of options for parallel simulations*, pp 325-332

[10] Chandy, K.M. and Misra J. [1979] *Distributed simulation: a Case Study in Design and Verification of Distributed Programs*, IEEE Transaction on Software engineering,5, pp 440-452

[11] Bryant, R.E. [1984] *A Switch-Level Model and Simulator for MOS Digital systems*, IEEE Transaction on Computer, 33(2), pp 160-177

[12] Bain, W.L. and Scot, D.S. [1988] *An Algorithm for Time Synchronisation in Distributed Discrete Event Simulation*, Proceedings of the SCS Multiconferenceon Distributed Simulation, 19, 3, pp 30-33

[13] Teo, Y.M. and Tay, S.C. [1994] *Algorithms for Conservative Parallel Simulation of Interconnection Networks*, Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks, pp 286-293

[14] Jefferson, D.R., [1985] , *Virtual Time* ACM Transaction on Programming Language System , pp404-425

[15] Jefferson, D.R. and Sowizral, H., [1982] , *Fast Concurrent Simulation Using the Time Wrap Mechanism* pp 77-93

[16] Stallings, W., [2000], Data and Computer Communications, Prencince Hall

[17] Wang, J. and Keshav, S. ,[1999] Efficient and Accurate Ehernet Simulation , Proceedings oh 24_{th} Conference on Local Computer Networks pp 182-201

Appendix A

Measurements

Problem size	Π_{prob}	Π_{ord}^{CMB}	Π_{sync}^{2PP}	Π_{sync}^{4PP}	Π_{sync}^{6PP}	Π_{sync}^{8PP}
24	1,26	3,50	1,35	2,16	2,60	2,96
48	1,38	3,82	1,30	2,07	2,50	2,85
72	1,40	4,70	1,30	2,02	2,46	2,73
96	1,49	4,87	1,29	2,03	2,48	2,78

Table A.1: Parallelism at different layers

Problem Size	Π_{prob}	Π_{ord}^{CMB}	$\Pi_{ord}^{Partial}$	Π_{ord}^{TI5}
24	1,26	3,50	20,07	7,21
48	1,38	3,82	40,04	14,36
72	1,40	4,70	60,792	21,02
96	1,49	4,87	81,03	27,32

Table A.2: Parallelism at simulation model layer

Problem size	M_{prob}	M_{ord}^{CMB}	$M_{ord}^{Partial}$	$M_{ord}^{TI(5)}$
24	192	772	1052,00	772
48	384	2312	3014,00	2307
72	576	5076	6234,00	5048
96	768	7369	9872,00	7329

Table A.3: Memory requirement at simulation model layer

Problem size	M_{prob}	M_{ord}^{CMB}	M_{sync}^{2PP}	M_{tot}^{2PP}	M_{sync}^{4PP}	M_{tot}^{4PP}	M_{sync}^{6PP}	M_{tot}^{6PP}	M_{sync}^{8PP}	M_{tot}^{8PP}
24	192	772	96	1060	408	1372	960	1924	1488	2452
48	384	2312	192	2888	768	3464	1968	4664	4544	7240
72	576	5076	288	5940	1224	6876	3312	8964	5472	11124
96	768	7369	384	8521	1572	9709	4416	12553	8832	16969

Table A.4: Memory requirement at different layers

Problem Size	S_{Prob}	S_{CMB}	S_{TI5}	$S_{partial}$	S_{2PP}	S_{4PP}	S_{6PP}	S_{8PP}
24	0,988	0,868	0,715	0,151	0,998	0,994	0,990	0,986
48	0,990	0,917	0,696	0,148	0,999	0,998	0,997	0,996
72	0,993	0,933	0,684	0,145	0,999	0,999	0,999	0,998
96	0,993	0,944	0,676	0,144	0,999	0,999	0,999	0,999

Table A.5: Strictness at different layers

Appendix B

SPaDES Manual Page

Example of configuration file for SPaDES/Java (CSMA example):

#CONFIGURATION FILE FOR SIMJAVA

#GENERAL CONFIGURATION

Algorithm: sequential

Stop(duration): 250000

Problem size: 96

Debugging log level: 1

Report file: reps/repCs96

Events file: yes

#PROBLEM SPECIFIC CONFIGURATION

Frame size: 64

Jam size: 4

Max buffer size: 8

Offered load: 1

Max propagation delay: 30

LAN speed: 10

```
SPADES(JAVA)                                                SPADES(JAVA)


NAME
       SPaDES/Java - a framework for discrete event simulation

DESCRIPTION
       SPaDES/Java  is  an  implementation  of the SPaDES (Structured Parallel
       Discrete-event Simulation) modeling and simulation framework  based  on
       the event-oriented modeling paradigm.

CONFIGURATION FILE
       SPaDES/Java optins can be set through a configuration file. The name of
       the file is given as a parameter to the simulator. If no configuration
       file is specified, the default configFile is assumed.

OPTIONS AVAILABLE IN THE CONFIGURATION FILE
       This are the options that can be set in the configuration file.  If any
       of the option is not specified, the default value is used.

       Algorithm: type
            select the simulation type, either sequential or parallel
             Default - sequential

       Stop(duration): number
            enable duration stopping condition
            Default: 10

       Stop(duration2): number
            enable duration stopping condition, allowing sheduled events  to
            execute
            Default: disabled

       Debugging log level: (0-4)
            sets the debugging level, 0 - less messages, 4 - all messages.
            Default: 4 - all messages

       Report file: filename
            sets the filename for the report file.
            Default: reportFile

       Events file: yes / no
            sets if event file is generated.
            Default: no

       Profile file: yes /no
            sets if profile file is generated.
            Default: no

       Hosts file: filename
            sets the filename for the hosts file. This file is used only for
            the parallel version of the simulator.  Please consult  examples
            for details of the syntax of this file.
            Default: hosts.txt

       Other options
```

Figure B.1: SPaDES/Java Man Page (1)

```
            Application specific options can be set also in this file.

Examples
       Configuration file:

         #CONFIGURATION FILE FOR SIMJAVA
         #GENERAL CONFIGURATION
         Algorithm: parallel
         Stop(duration) : 100
         Debugging log level: 4
         Report file: repPar
         Event file: yes
         Hosts file: hosts.txt
         ===================
         #PROBLEM SPECIFIC CONFIGURATION
         Row size: 2
         Column size: 2
         Messages per server: 1
         Mean service time: 1
         Transport time: 1
         ===================
       The syntax for a line in the configuration file is: "Option : value"


       Hosts file:
         0 63 1 compute-0-1.ddns.comp.nus.edu.sg
         64 127 1 compute-0-2.ddns.comp.nus.edu.sg
         128 191 1 compute-0-3.ddns.comp.nus.edu.sg
         192 255 1 compute-0-4.ddns.comp.nus.edu.sg

       This  means  that  there      are  255  logical processes, the first 64 are
       mapped on the first host and so on. The third parameter    on  each  line
       represents  the  step. Step  =      1  means that all logical processes 0
       through 63 are mapped on the firs host and so on.

       Another example:
         0 127 2 compute-0-1.ddns.comp.nus.edu.sg
         1 127 2 compute-0-2.ddns.comp.nus.edu.sg
         128 255 2 compute-0-3.ddns.comp.nus.edu.sg
         129 255 2 compute-0-4.ddns.comp.nus.edu.sg

       Step = 2 means that logical process 0,2,4, etc are mapped on the  first
       host, logical processes 1,3,5, etc are mapped on the second host and so
       on.


WRITING A NEW APPLICATION
       For writing a new application, SPaDES/Java comes with 3 example  appli-
       cations, PHOLD and MIN and CSMA

AUTHOR
       The SPaDES/Java version 0.2 is freeware.



May 2005                     0.2                 SPADES(JAVA)
```

Figure B.2: SPaDES/Java Man Page (2)